Konfigurierbare Systemsoftware (KSS)

VL 1 – Einführung

Daniel Lohmann
Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme
Friedrich-Alexander-Universität
Erlangen-Nürnberg

SS 12 – 18. April 11
http://www4.informatik.uni-erlangen.de/Lehre/SS12/V_KSS

Agenda

1.1 Commodity Operating Systems Today
1.2 Reality Check: Granularity
1.3 The Domain of Embedded Systems
1.4 About KSS
1.5 KSS — Organization
1.6 References

The Operating System – A Swiss Army Knife?

Commodity operating systems provide a rich set of features to be prepared for all kinds of applications and contingencies:

- Malicious or erroneous applications
  - preemptive scheduling, address space separation, disk quotas
- Multi-user operation
  - authentication, access validation and auditing
- Multi-threaded and interacting applications
  - Threads, semaphores, pipes, sockets
- Many/large concurrently running applications
  - virtual memory, swapping, working sets

One size fits all?

"Clearly, the operating system design must be strongly influenced by the type of use for which the machine is intended. Unfortunately it is often the case with 'general purpose machines' that the type of use cannot be easily identified; a common criticism of many systems is that in attempting to be all things to all men they wind up being totally satisfactory to no-one."

Some applications may require only a subset of services or features that other applications need. These 'less demanding' applications should not be forced to pay for the resources consumed by unneeded features.

Parnas 1979: “Designing Software for Ease of Extension and Contraction” [8]

Variability (Definition 1)
Variability of system software is the property that denotes the range of functional requirements that can be fulfilled by it.

Granularity (Definition 2)
Granularity of system software is the property that denotes the resolution of which requirements can be fulfilled by it, in the sense that requirements are fulfilled but not overfulfilled.

Can general purpose (GP) systems fulfill these demands?

Reality check – a small study with `printf()` from glibc:
(Analogy: GP operating system ←→ GP library ←→ GP function)

```
int main() {
    printf( "Hello World\n");
}
```

Maybe the general-purpose `printf()` is just too powerful?
- supports many data types, formatting rules, ...
- implementation requires a complex parser for the format string

Let’s try the much more specialized `puts()`!
Experiment 2: `puts()`

```bash
> echo 'main(){puts("Hello World");}' | gcc -xc -Os -w -static -o hello2
> ./hello2
Hello World
> size hello2
text data bss dec hex filename
508723 1928 7522 517703 7e47 hello2
```

512 KiB!

That didn’t help much!
- Maybe `puts()` is yet too overpowerful?
- buffered IO, streams
- Let’s work directly with the OS file handle!

Experiment 3: `write()`

```bash
> echo 'main(){write(1, "Hello World\n", 13);} | gcc -xc -Os -w -static -o hello3
> ./hello3
Hello World
> size hello3
text data bss dec hex filename
508138 1928 7052 517118 7e3fe hello3
```

512 KiB!

517703 compared to 517118 – a net saving of **585 bytes (0.1%)**!:-(

Experiment 4: empty program

```bash
> echo 'main(){} | gcc -xc -Os -w -static -o hello4
> size hello4
text data bss dec hex filename
508074 1928 7052 517054 7e3be hello4
```

Hm...

```bash
objdump -D --reloc hello4 | grep printf | wc -l
```

yields still **2611 matches!**

It’s the startup code!

Experiment 5: `write()`, no startup code

```bash
> echo '_start(){write(1, "Hello World\n", 13);_exit(0);} | gcc -xc -Os -w -static -nostartfiles -o hello5
> size hello5
text data bss dec hex filename
597 0 4 601 259 hello5
```

0.5 KiB :-|

but segfault :-(

Segmentation fault

Even a simple `write()` cannot be issued without the complete initialization.

Last ressort: invoke the syscall directly!

Experiment 6: `SYS_write()`

```bash
> echo '_start(){syscall(4, 1, "Hello World\n", 13);_exit(0);} | gcc -xc -Os -w -static -nostartfiles -o hello6
> size hello6
text data bss dec hex filename
293 0 4 297 129 hello6
```

0.25 KiB :-)

On Linux/glibc, a simple “Hello World” application takes **1750 times** more memory than necessary!

However, is this a problem?
- The glibc has been designed for a “standard case”
  - Large, multithreaded, IO-intensive UNIX application
  - Assumption: every program uses `malloc()`, `printf()`, ...

Variability has been traded for Granularity

Every Program?

> “I know of no feature that is always needed. When we say that two functions are always almost used together, we should remember that “almost” is a euphemism for “not”.”

Parnas 1979: “Designing Software for Ease of Extension and Contraction” [8]
Reality Check: Lessons Learned

On Linux/glibc, a simple "Hello World" application takes 1750 times more memory than necessary!

However, is this a problem?
- The glibc has been designed for a "standard case":
  - Large, multithreaded, IO-intensive UNIX application
  - Assumption: every program uses malloc(), printf(), ...
- Variability has been traded for Granularity
- Assumption: The GP operating system will compensate for it...

Virtual memory ~ memory is not an issue (but is that a reason to waste it?)
- Shared libraries ~ memory is actually shared between processes (unless we relocate the symbols, e.g., for address-space randomization...)

Variability has been traded for Granularity
Assumption: The GP operating system will compensate for it...

What about other domains?

A Different Domain: Embedded Systems

CPU/DSP sales in 2002 [13]

Limited Resources
- Flash is limited, RAM is extremely limited
- A few bytes can have a massive impact on per-unit costs
- The "glibc approach" is doomed to fail!
The Role of the Operating System

Operating systems (not) employed in embedded-system projects in 2006 [12]

> 40% of all projects use “in house” OS functionality!

Wide-spread fear of the resource overhead of GP operating systems

- OS functionality is developed “side-by-side” with the applications
- This leads to very high “hidden” development costs [14]

> Rest spreads over hundreds of different operating systems!

- (C1, 166, 251), Ciao, CMX RTOS, Contiki, C-Smart/Phozen, eCos, RTOS, Embera, Entice, Eureka Plus, FreesRTOS, H: Ren, HybridOS, LynxOS, MicroX/OS-II, Nucleos, OS-9, OSE, OSEK (Flex, Turbo, Plus), OSEK (Linux, Precise/MQX, Precise/RTOS, proOSEK), pSOS, PURE, PARCOS, QNX, Realtime, RT-Motion, RTA, RTX/100, 2511), RTXC, Softimage, SSSX RTOS, ThreadOS, TinyOS, Tracex, VRTEX, VxWorks, ...

- The “glibc approach” (one size fits all) does not work!
Between a Rock and a Hard Place...

- High variety of functional and nonfunctional requirements
- High variety of hardware platforms
- High per-unit cost pressure

~ System software has to be tailored for each concrete application

Agenda

1.1 Commodity Operating Systems Today
1.2 Reality Check: Granularity
1.3 The Domain of Embedded Systems
1.4 About KSS
1.5 KSS — Organization
1.6 References

Customizing / Tailoring

Customizing or tailoring is the activity of modifying existing system software in order to fulfill the requirements of some particular application.

This calls for granularity and variability!

Functional and nonfunctional requirements

- tasks
- sockets
- file system
- event latency
- safety
- ISA
- IRQ handling
- MMU / MPU
- cache size
- coherence
- IRQ latency

Functional and nonfunctional properties

- High variety of functional and nonfunctional application requirements
- High variety of hardware platforms
- High per-unit cost pressure

~ System software has to be tailored for each concrete application
What to do?

297 $\leftrightarrow$ 517703 Bytes!

Why?

On Linux/glibc, a simple “Hello World” application takes 1750 times more memory than necessary!

- Reason: software structure
  - Trade-off between reuse $\iff$ coupling
    $\leadsto$ by extensive internal reuse, glibc has become an all-or-nothing blob
- Reason: software interface
  - C standard defines printf() as a swiss army knife [3, §7.19.6]
    $\leadsto$ printf() has become a “god method” [1]
- Reason: language and tool chain
  - Compiler/linker work on the granularity of symbols or even object files
    $\leadsto$ dead code is not effectively eliminated

---

Individually Developed Software Product

Software Product Derived from Reusable Assets

Konfigurierbare Systemsoftware – KSS

Throughout the software development cycle, variability and granularity have to be considered as primary design goals from the very beginning!

In KSS you will learn about principles, methods, and tools to achieve this.
Agenda

1.1 Commodity Operating Systems Today
1.2 Reality Check: Granularity
1.3 The Domain of Embedded Systems
1.4 About KSS
1.5 KSS — Organization
1.6 References

Learning Objectives

- **Improve** your understanding of the design and development of low-level system software
  - Starting point: “Betriebssysteme” [BS]
  - Focus: Static configuration and tailoring

- **Expand** your knowledge by new software engineering methods and language techniques for configurable system software
  - Software families and software product lines [7]
  - Aspect-oriented and generative programming in C/C++ [10]

- **Apply** these techniques in the context of current operating-system research projects
  - CiAO, Sloth, VAMOS, DanceOS [2, 5, 9, 11]
  - Get prepared for a master thesis or project in the field!
## Organisation: Systemsoftwaretechnik (SST)

- **Modul Systemsoftwaretechnik (SST)** 7.5 ECTS
  - **1. Vorlesung Betriebssystemtechnik (BST)** 2.5 ECTS
    - Di 10–12
    - 12–14 Vorlesungstermine
  - **2. Übungen zu Betriebssystemtechnik (BST-Ü)** 2.5 ECTS
    - Mi 16–18 oder Do 12–14
    - 12–14 Übungstermine + Rechnerübungen
  - **3. Vorlesung und Übung Konfigurierbare Systemsoftware (KSS)** 2.5 ECTS
    - Mi 10–12 (Vorlesung)
    - 7 Vorlesungstermine, 2 Übungsaufgaben, 1 Projekt
    - Übung integriert in BST-Übung / Rechnerübung

- ~ KSS kann **nur zusammen mit BST** belegt werden!
- Es gibt keine 2.5 ECTS Module...
- Ab 2013 wird KSS vermutlich auf 5 ECTS haben